Reductions for Synthesis Procedures*

Swen Jacobs[1], Viktor Kuncak[2], and Philippe Suter[2]

[1] Graz University of Technology, Austria
swen.jacobs@iaik.tugraz.at
[2] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
firstname.lastname@epfl.ch

Abstract. A synthesis procedure acts as a compiler for declarative specifications. It accepts a formula describing a relation between inputs and outputs, and generates a function implementing this relation. This paper presents the first synthesis procedures for 1) algebraic data types and 2) arrays. Our procedures are reductions that lift a synthesis procedure for the elements into synthesis procedures for containers storing these elements. We introduce a framework to describe synthesis procedures as systematic applications of inference rules. We show that, by interpreting both synthesis problems and programs as relations, we can derive and modularly prove widely applicable transformation rules, simplifying both the presentation and the correctness argument.

1 Introduction

Software synthesis is an active area of research [6, 17, 19, 22]. It has received increased attention recently, but has been studied for decades [3, 11, 12, 16]. Our paper pursues the synthesis of functions mapping inputs to outputs. The synthesized functions are guaranteed to satisfy a given input/output relation expressed in a decidable logic. We call this approach complete functional synthesis [8, 9]. The appeal of this direction is that it can synthesize functions over unbounded domains, and that the produced code is guaranteed to satisfy the specification for the entire unbounded range of inputs. If the synthesis process always terminates, we speak of *synthesis procedures*, analogously to decision procedures.

Previous work described synthesis procedures for linear arithmetic and sets [8, 9] as well as extensions to unbounded bitvector constraints [4, 18]. In this paper we make further steps towards systematic derivation of synthesis procedures by showing how inference rules that describe decision procedure steps (possibly for a combination of theories) can be generalized to synthesis procedures. Within this framework we derive the first synthesis procedures for two relevant decidable theories of data structures: term algebras (algebraic data types), and the

* Swen Jacobs was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613), the Austrian Science Fund (FWF) through the national research network RiSE (S11406), the Swiss NSF Grant 200021_132176, and COST Action IC0901. Philippe Suter was supported in part by the Swiss NSF Grant 200021_120433.

R. Giacobazzi, J. Berdine, and I. Mastroeni (Eds.): VMCAI 2013, LNCS 7737, pp. 88–107, 2013.

theory of integer-indexed arrays with symbolic bounds on index ranges. The two synthesis procedures that we present are interesting in their own right. Synthesis for algebraic data types can be viewed as a generalization of the compilation of pattern matching, and is therefore a useful way to increase the expressive power of functional programs. Synthesis for arrays is useful for synthesizing fragments of imperative programs. Synthesizing from constraints on arrays is challenging because it requires, in general, iteration over the space of indices. It therefore illustrates the importance of synthesizing not only individual values that meet a constraint, but also functions that enumerate all values.

Our synthesis procedures are expressed as a set of modular transformation rules whose correctness can be checked in a straightforward way, and which can be more easily implemented (even in foundational proof assistants). The transformations gradually evolve a constraint into a program. Sound rules for such transformations can be formulated for each decidable theory separately, and they can be interleaved for more efficient synthesis and more efficient synthesized programs. Our framework therefore contributes to the methodology for synthesis in general. We start from proof rules for a decision procedure, and extend them into transformation rules that can be viewed as a result of partially evaluating the execution of inference rules.

As remarked in [9], compiled synthesis procedures could be viewed as a result of partial evaluation of the execution of a constraint solver at run time. This is a useful observation from a methodological point of view. However, it likely has similar limitations as an attempt to automatically transform an interpreter into a compiler. We therefore expect that the insights of researchers will continue to play a key role in designing synthesis procedures. These insights both take the form of understanding decidable logics, but also understanding how to solve certain classes of problems efficiently. Examples of manually deriving compiled code that can be more efficient than run-time search appear in both synthesis for term algebras and the synthesis of arrays. We can assume that the values in these theories are finitely generated by terms. Because these terms become known only at run time, it appears, at first, necessary to continue running decision procedure at run time. However, because the nature of processing steps is known at compile time, it was possible to generate statically known loops instead of an invocation of a general-purpose constraint solver at run time. The main advantage is not only that such code can be more efficient by itself, but that it can then be further analyzed and simplified, automatically or manually, to obtain code that is close or better than one written using conventional development methodology.

Contributions. In summary, this paper makes the following contributions:

1. the first synthesis procedure for quantifier-free theory of algebraic data types;
2. the first synthesis procedure for a theory of (symbolically bounded) arrays;
3. a formalization of the above procedures, as well as a simple synthesis procedure for Presburger arithmetic, in a unified framework supporting:
   (a) proving correctness of synthesis steps, and
   (b) combining synthesis procedures in a sound way.

We start by introducing our framework and illustrate it with a simple synthesis procedure for Presburger arithmetic. We then present the synthesis procedures for algebraic data types and for arrays.

2 Synthesis Using Relation Transformations

A *synthesis problem* is a triple

$$[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!]$$

where $\bar{a}$ is a set of *input* variables, $\bar{x}$ is a set of *output* variables and ϕ is a formula whose free variables are a subset of $\bar{a} \cup \bar{x}$. A synthesis problem denotes a binary relation $\{(\bar{a}, \bar{x}) \mid \phi\}$ between inputs and outputs. The goal of synthesis is to transform such relations until they become executable programs. Programs correspond to formulas of the form $P \wedge (\bar{x} = \bar{T})$ where $\mathrm{vars}(P) \cup \mathrm{vars}(\bar{T}) \subseteq \bar{a}$. We denote programs

$$\langle P \mid \bar{T} \rangle$$

We call the formula P a *precondition* and call the term $\bar{T}$ a *program term*.

We use $\vdash$ to denote the transformation on synthesis problems, so

$$[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!] \vdash [\![\bar{a}\ \langle\phi'\rangle\ \bar{x}]\!] \tag{1}$$

means that the problem $[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!]$ can be transformed into the problem $[\![\bar{a}\ \langle\phi'\rangle\ \bar{x}]\!]$. The variables on the right-hand side are always the same as on the left-hand side. Our goal is to compute, given $\bar{a}$, one value of $\bar{x}$ that satisfies ϕ. We therefore define the soundness of (1) as a process that refines the binary relation given by ϕ into a smaller relation given by ϕ', without reducing its domain. Expressed in terms of formulas, the conditions become the following:

$$\begin{array}{rl} \phi' \models \phi & \textit{refinement} \\ \exists \bar{x}.\phi \models \exists \bar{x}.\phi' & \textit{domain preservation} \end{array}$$

In other words, $\vdash$ denotes domain-preserving refinement of relations. Note that the dual entailment $\exists \bar{x}.\phi' \models \exists \bar{x}.\phi$ also holds, but it follows from *refinement*. Note as well that $\vdash$ is transitive.

Equivalences in the theory of interest immediately yield useful transformation rules: if ϕ and ϕ' are equivalent, (1) is sound. We can express fact as the following *inference rule*:

$$\frac{\models \phi_1 \leftrightarrow \phi_2}{[\![\bar{a}\ \langle\phi_1\rangle\ \bar{x}]\!] \vdash [\![\bar{a}\ \langle\phi_2\rangle\ \bar{x}]\!]} \tag{2}$$

In most cases we will consider transformations whose result is a program:

$$[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle$$

The correctness of such transformations reduces to

$$\begin{array}{rl} P \models \phi[\bar{x} \mapsto \bar{T}] & \textit{refinement} \\ \exists \bar{x}.\phi \models P & \textit{domain preservation} \end{array}$$

A *synthesis procedure* for a theory $\mathcal{T}$ is given by a set of inference rules and a strategy for applying them such that every formula in the theory is transformed into a program.

2.1 Theory-Independent Inference Rules

We next introduce inference rules for a logic with equality. These rules are generally useful and are not restricted to a particular theory.

Equivalence. From the transitivity of $\vdash$ and (2), we can derive a rule for synthesizing programs from equivalent predicates.

$$\frac{[\![\bar{a}\ \langle\phi_1\rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle \qquad \models \phi_1 \leftrightarrow \phi_2}{[\![\bar{a}\ \langle\phi_2\rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle}$$

Ground. In the case where no input variables are given, a synthesis problem is simply a satisfiability problem.

$$\frac{\mathcal{M} \models \phi}{[\![\emptyset\ \langle\phi\rangle\ \bar{x}]\!] \vdash \langle \top \mid \mathcal{M}\rangle} \qquad \frac{\neg\exists\mathcal{M}.\mathcal{M} \models \phi}{[\![\emptyset\ \langle\phi\rangle\ \bar{x}]\!] \vdash \langle \bot \mid \bot\rangle}$$

(In these rules $\mathcal{M}$ is a *model* for ϕ and should be thought of as a tuple of ground terms.) Note that the second rule can be generalized: even in the presence of input variables, if the synthesis predicate ϕ is unsatisfiable, then the generated program must be $\langle \bot \mid \bot\rangle$.

Assertions. Parts of a formula that only refer to input variables are essentially assertions and can be moved to the precondition.

$$\frac{[\![\bar{a}\ \langle\phi_1\rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle \qquad \mathrm{vars}(\phi_2) \subseteq \bar{a}}{[\![\bar{a}\ \langle\phi_1 \wedge \phi_2\rangle\ \bar{x}]\!] \vdash \langle \phi_2 \wedge P \mid \bar{T}\rangle}$$

Case Split. A top-level disjunction in the formula can be handled by deriving programs for both disjuncts and combining them with an if-then-else structure.

$$\frac{[\![\bar{a}\ \langle\phi_1\rangle\ \bar{x}]\!] \vdash \langle P_1 \mid \bar{T}_1\rangle \qquad [\![\bar{a}\ \langle\phi_2\rangle\ \bar{x}]\!] \vdash \langle P_2 \mid \bar{T}_2\rangle}{[\![\bar{a}\ \langle\phi_1 \vee \phi_2\rangle\ \bar{x}]\!] \vdash \langle P_1 \vee P_2 \mid \mathsf{if}(P_1)\ \{\bar{T}_1\}\ \mathsf{else}\ \{\bar{T}_2\}\rangle}$$

Unconstrained Output. Output variables that are not constrained by ϕ can be assigned any value.

$$\frac{[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T}\rangle \qquad x_0 \notin \mathrm{vars}(\phi)}{[\![\bar{a}\ \langle\phi\rangle\ x_0\,;\bar{x}]\!] \vdash \langle P \mid \mathsf{any}\,;\bar{T}\rangle}$$

In the program, any denotes a nullary function that returns an arbitrary of the appropriate type.

One-point. Whenever the value of an output variable is uniquely determined by an equality atom, it can be eliminated by a simple substitution.

$$\frac{[\![\bar{a}\ \ \langle \phi[x_0 \mapsto t] \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad x_0 \notin \mathrm{vars}(t)}{[\![\bar{a}\ \ \langle x_0 = t \wedge \phi \rangle\ \ x_0 \,;\, \bar{x}]\!] \vdash \langle P \mid \mathsf{let}\ \bar{x} := \bar{T}\ \mathsf{in}\ (t \,;\, \bar{x}) \rangle}$$

Definition. The definition rule is in a sense dual to One-point, and is convenient to give a name to a subterm appearing in a formula. Typical applications include purification and flattening of terms.

$$\frac{[\![\bar{a}\ \ \langle x_0 = t \wedge \phi[t \mapsto x_0] \rangle\ \ x_0 \,;\, \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad x_0 \notin \mathrm{vars}(\phi)}{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \mathsf{let}\ (x_0 \,;\, \bar{x}) := \bar{T}\ \mathsf{in}\ \bar{x} \rangle}$$

Sequencing. The sequencing rule allows us to synthesize values for two groups of variables one after another. It fixes the values of some of the output variables, treating them temporarily as inputs, and then continues with the synthesis of the remaining ones.

$$\frac{[\![\bar{a} \,;\, \bar{x}\ \ \langle \phi \rangle\ \ \bar{y}]\!] \vdash \langle P_1 \mid \bar{T}_1 \rangle \qquad [\![\bar{a}\ \ \langle P_1 \rangle\ \ \bar{x}]\!] \vdash \langle P_2 \mid \bar{T}_2 \rangle}{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x} \,;\, \bar{y}]\!] \vdash \langle P_2 \mid \mathsf{let}\ \bar{x} := \bar{T}_2\ \mathsf{in}\ (\bar{x} \,;\, \bar{T}_1) \rangle}$$

Static Computation. A basic rule is to perform computational steps when possible.

$$\frac{[\![a_0 \,;\, \bar{a}\ \ \langle \phi[t \mapsto a_0] \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad \mathrm{vars}(t) \subseteq \bar{a} \qquad a_0 \notin \mathrm{vars}(\phi)}{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x}]\!] \vdash \langle \mathsf{let}\ a_0 := t\ \mathsf{in}\ P \mid \mathsf{let}\ a_0 := t\ \mathsf{in}\ \bar{T} \rangle}$$

Variable Transformation. The $\vdash$ transformation preserves the variables. To show how we can change the set of variables soundly, we next present in our framework *variable transformation* by a computable function ρ [8], as an inference rule on two $\vdash$ transformations.

$$\frac{[\![\bar{a}\ \ \langle \phi[\bar{x} \mapsto \rho(\bar{x}')] \rangle\ \ \bar{x}']\!] \vdash \langle P \mid \bar{T} \rangle}{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \rho(\bar{T}) \rangle}$$

Slightly more generally, we have the following:

$$\frac{[\![\bar{a}\ \ \langle \phi' \rangle\ \ \bar{x}']\!] \vdash \langle P \mid \bar{T} \rangle \qquad \exists \bar{x}.\phi \models \exists \bar{x}'.\phi' \qquad \phi' \models \phi[\bar{x} \mapsto \rho(\bar{x}')]}{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \rho(\bar{T}) \rangle}$$

Existential Projection. This rule is a special case of variable transformation, where ρ simply projects out some of the variables.

$$\frac{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x} \,;\, \bar{x}']\!] \vdash \langle P \mid \bar{T} \rangle}{[\![\bar{a}\ \ \langle \exists \bar{x}'.\phi \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \mathsf{let}\ (\bar{x} \,;\, \bar{x}') := \bar{T}\ \mathsf{in}\ \bar{x} \rangle}$$

3 Synthesis for Presburger Arithmetic

This section summarizes a simple version of a synthesis procedure for Presburger arithmetic using our current synthesis rules. Our goal is to give a complete procedure that is easy to prove correct, as opposed to one that generates efficient code. The reader will observe that our description reads like a description of quantifier elimination. Note, however, that the inference rules that we refer to are from the previous section and therefore also specify how to compute the corresponding program.

Unlike the procedure in [9] the procedure below does not perform efficient solving of equations, but could be refined to do so by adapting the description in [9] to our inference rules.

As in the preprocessing steps for simple quantifier elimination for Presburger arithmetic, the equivalences we use as rules include replacing $t_1 \neq t_2$ with $t_1 < t_2 \vee t_2 < t_1$. In principle, we can rewrite $t_1 = t_2$ into $t_1 \leq t_2 \wedge t_2 \leq t_1$ (see [9] for more efficient approaches). We rewrite $t_1 \leq t_2$ into $t_1 < t_2 + 1$. When needed, we assume that we apply the Case Split rule to obtain only a conjunction of literals. We also assume that we apply the Sequencing rule to fix the remaining variables and only consider one output variable x. Finally, thanks to the Assertions rule, we assume that all literals contain x.

A rule that takes into account divisibility is the following:

$$\frac{[\![\bar{a}\ \langle \phi[kx \mapsto y] \wedge y \equiv_k 0 \rangle\ y]\!] \vdash \langle P \mid T \rangle \qquad k \neq 0 \qquad x \text{ in } \phi \text{ only as } kx}{[\![\bar{a}\ \langle \phi \rangle\ x]\!] \vdash \langle P \mid T/k \rangle}$$

The rule is a case of Variable Transformation with $\rho(y) = y/k$.

To enable the previous rule, we can ensure that all occurrences of a variable have the same coefficient by multiplying constraints by a positive constant (e.g., the least common multiple of all coefficients). These transformations are based on using (in a context) equivalences between $t_1 \bowtie t_2$ and $kt_1 \bowtie kt_2$, for $k > 0$ and $\bowtie \in \{=, <, >, \equiv_p\}$.

Using the rules so far, we can ensure that an output variable has a coefficient 1. If such a variable occurs in at least one equality, we can eliminate it using One-point. If the variable occurs only in inequalities, we perform the main step of the procedure.

Elimination of Inequalities. Based on the discussion above, we can assume that the formula ϕ in the synthesis problem is of the form

$$\bigwedge_{i=1}^{L} l_i < x \wedge \bigwedge_{j=1}^{U} x < u_j \wedge \bigwedge_{i=1}^{D} x + t_i \equiv_{K_i} 0$$

We aim to replace ϕ with ϕ' such that

$$[\![\bar{a}\ \langle \phi \rangle\ x]\!] \vdash [\![\bar{a}\ \langle \phi' \rangle\ x]\!] \tag{3}$$

We define ϕ' as

$$\bigvee_{i=1}^{L} \bigvee_{k=1}^{K} (\phi \wedge x = l_i + k)$$

where K is the least common multiple of $K_1, \ldots, K_D$. Clearly ϕ' is stronger than ϕ because each disjunct is stronger than ϕ, so it remains to argue about domain preservation. Suppose there exists values for x so that ϕ holds. Let l_I be the largest value among the values of lower bounds l_i and let T be such that $l_I + T \equiv_K x$ holds. Then letting x to be $l_I + T$ makes ϕ' true as well.

After performing disjunctive splitting (Case Split), we can eliminate x using the One-point rule. The correctness follows by (3) and the correctness of One-point. The cases where some of the bounds do not exist can be treated similarly.

This completes the overview of synthesis of functions given by Presburger arithmetic relations.

Enumerating Solutions. In addition to finding one solution $\bar{x}$ such that ϕ holds, it is useful to be able to find all solutions, when this set is finite. When solving constraints at run time, a simple way to find all solutions is to maintain a list of previously found solutions $\bar{v}_1, \ldots, \bar{v}_n$ for $\bar{x}$ and add to ϕ an additional conjunct $\bigwedge_{i=1}^{n} \bar{x} \neq \bar{v}_n$, see [7].

One possible approach to compile this process is to enrich Presburger arithmetic with finite uninterpreted relations as parameters. This enables a synthesis procedure to, for example, take the set of previous solutions as the input. If R is such a finite-relation symbol or arity n and $\bar{x}$ are n variables, we introduce an additional literal $\bar{x} \notin R$ into the logic, with the intention that R stores the previously found solutions. The elimination of inequalities then produces terms that avoid the elements of R by considering not only the value $l_i + k$ for x, but enumerating a larger number of solutions, $l_i + k + \alpha N$, for multiple values of $\alpha \geq 0$. Because R is known only at run time, the generated code contains a loop that increases $\alpha \geq 0$ to allow x to leave the range of the corresponding coordinate of R. The value of α is bounded at run time by, for example, $\lceil (\max(R_x) - \min(R_x))/K \rceil + 1$ where R_x is the projection of R onto the coordinate at which x appears in the literal $\bar{x} \notin R$. The generated loop is guaranteed to terminate.

4 Synthesis for Term Algebras

This section presents a synthesis procedure for quantifier-free formulas in the theory of term algebras. We start by assuming a pure term algebra, and later extend the system to algebras with elements from parameter theories. In both cases, we present a series of normal forms and inference rules, and argue that together they can be used to build a synthesis procedure.

4.1 Pure Term Algebras

The grammar of atoms over our term algebra is given by the following two production rules, where c and F denote a constant and a function symbol from the algebraic signature, respectively:

$$\begin{aligned} A &::= T = T \mid T \neq T \mid \mathrm{is}_c(T) \mid \mathrm{is}_F(T) \\ T &::= x \mid c \mid F(\bar{T}) \mid F_i(T) \end{aligned}$$

In the following we assume that the algebra defines at least one constant and one non-nullary constructor function. Formulas are built from atoms with the usual propositional connectives. We use an extension of the standard theory of term algebras. The extension defines additional unary *tester* functions $\mathrm{is}_c(\cdot)$ and $\mathrm{is}_F(\cdot)$ for constant and functions in the algebraic signature respectively, and unary *selector* functions $F_i(\cdot)$, with $1 \leq i \leq n$ where n is the arity of F. These extra symbols form a definitional extension [5] given by the axioms:

$$\forall x.\mathrm{is}_c(x) \leftrightarrow x = c \tag{4}$$

$$\forall x.\mathrm{is}_F(x) \leftrightarrow \exists \bar{y}.x = F(\bar{y}) \tag{5}$$

$$\begin{aligned} \forall x, y.F_i(x) = y \leftrightarrow (\exists \bar{y}.y = \bar{y}\,[i] \wedge F(\bar{y}) = x) \\ \vee \neg(\exists \bar{y}.F(\bar{y}) = x) \wedge x = y \end{aligned} \tag{6}$$

Note that the case analysis in (6) is required only to make the selector functions total. In practice, we are only interested in cases where the selectors are applied to arguments of the proper type. We will therefore assume in the following that each selector application $F_i(x)$ is accompanied with a side condition $\mathrm{is}_F(x)$.

Rewriting of tester and selector functions. By applying the axioms (4) and (5), we can rewrite all applications of a tester function into an existentially quantified equality over terms. We can similarly eliminate applications of testers by existentially quantifying over the arguments of the corresponding constructors. Using the Existential Projection rule, we can in turn consider the obtained synthesis problem as a quantifier-free one.

Elimination through unification. We can at any point apply *unification* to a conjunction of equalities over terms. Unification rewrites a conjunction of term equations into either $\bot$, if the equations contain a cycle or an equality involving incompatible constructors, or into an equivalent conjunction of atoms $\bigwedge_i v_i = t_i$, where v_i is a variable and t_i is a term. This set of equations has the additional property that

$$\left(\bigcup_i \{v_i\}\right) \cap \left(\bigcup_i \mathrm{vars}(t_i)\right) = \emptyset$$

In other words, it defines a set of variables as a set of terms built from another disjoint set of variables [2]. This form is particularly suitable for applications of the One-point rule: indeed, whenever v_i is an output variable, we can apply it, knowing that v_i does not appear in t_i (or in any other equation).

Dual view. Unification allows us to eliminate output variables that are to the left of an equality. When instead an input variable appears in such position, we can resort to a dual form to eliminate output variables appearing in the right-hand side. We obtain the dual form by applying as much as possible the following two rules to term equalities:

$$\frac{t = c}{\mathrm{is}_c(t)} \qquad \frac{t = F(t_1, \ldots, t_n)}{F_1(t) = t_1 \wedge \ldots \wedge F_n(t) = t_n \wedge \mathrm{is}_F(t)}$$

Note that these are rewrite rules for formulas. Because they preserve the set of variables and equisatisfiability, they can be lifted to inference rules for programs using the Equivalence rule. Observe that at saturation, the generated atoms are of two kinds: 1) applications of tester predicates and 2) equalities between two terms, each containing at most one variable. In particular, all equalities between an output variable and a term are amenable to applications of the One-point rule.

Disequalities. Finally, we introduce a dedicated rule for the treatment of disequalities between terms. The rule is defined for disequalities over variables and constants in *conjunctive normal form* (CNF). From a conjunction of disequalities over terms, we can obtain CNF by applying the following rewrite rules until saturation:

$$\frac{F(\bar{t}_1) \neq G(\bar{t}_2) \qquad F \neq G}{\top} \qquad \frac{F(\bar{t}_1) \neq F(\bar{t}_2)}{t_1^1 \neq t_1^n \vee \ldots \vee t_2^1 \neq t_2^n}$$

Intuitively, the first rule captures the fact that terms built with distinct constructors are trivially distinct (note that this also captures distinct constants, which are nullary constructors). The second rule breaks down a disequality into a disjunction of disequalities over subterms.

To obtain witness terms from the CNF, it suffices to satisfy one disequality in each conjunct. We achieve this by eliminating one variable after another, applying for each a diagonalization principle, as follows. In the following rule ϕ_{CNF} denotes the part of the CNF formula over atomic disequalities that does not contain a given variable of interest x_0.

$$\frac{[\![\bar{a}\ \langle \phi_{\mathrm{CNF}} \rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle \qquad x_0 \notin \phi_{\mathrm{CNF}}}{\left[\!\!\left[\bar{a} \left\langle \begin{array}{c} (x_0 \neq t_1 \vee \ldots) \\ \wedge \quad \ldots \\ \wedge\ (x_0 \neq t_n \vee \ldots) \\ \wedge \quad \phi_{\mathrm{CNF}} \end{array} \right\rangle x_0 \,;\, \bar{x} \right]\!\!\right] \vdash \langle P \mid \mathsf{let}\ \bar{x} := \bar{T}\ \mathsf{in}\ (\Delta(t_1, \ldots, t_n)\,;\, \bar{T}) \rangle}$$

In the generated program, Δ denotes an n-ary computable function that returns, at run time, a term distinct from all its arguments. Such a value is guaranteed to exist, since the term algebra is assumed to have at least one constructor. This function runs in time polynomial in the number of its arguments.

Synthesis Procedure for Algebraic Data Types. We now argue that the reductions to normal forms and the rules presented above are sufficient to form a complete synthesis procedure for any given pure term algebra structure. The procedure (a strategy for applying the rules) is given by the following steps:

1. Reduce an arbitrary propositional structure to a conjunction through applications of the Case Split rule.
2. Remove selectors and testers through rewrites and applications of Existential Projection.
3. Apply unification to all equalities, then apply One-point as often as possible. As a result, the only equalities remaining have the form $a = t$, where a is an input variable and $a \notin \text{vars}(t)$.
4. Rewrite into dual form, then apply One-point as much as possible. After applying Assertions, the problem is reduced to a conjunction of disequalities, each involving at least one output variable.
5. Transform the conjunction into CNF and eliminate all remaining variables by successive applications of the diagonalization rule.

Given a conjunctions of literals, the generated program runs in time polynomial in the size of the input terms: it consists of a sequence of assignments, one for each output variable, and each term has polynomial size.

4.2 Reduction to an Interpreted Theory

We now consider the case of a term algebra defined over an interpreted theory $\mathcal{T}$. A canonical example is the algebra of integer lists, where $\mathcal{T}$ is the theory of integers, and defined by the constant $\mathsf{Nil} : \mathsf{List}$ and the constructor $\mathsf{Cons} : \mathbb{Z} \times \mathsf{List} \to \mathsf{List}$. In this theory, the selector function $\mathsf{Cons}_1(\cdot)$, for instance, is of type $\mathsf{List} \to \mathbb{Z}$. We show how to reduce a synthesis problem in the combination of theories to a synthesis problem in $\mathcal{T}$. We focus on the important differences with the previous case.

Purification. We can assume without loss of generality that constructor terms contain no subterms from $\mathcal{T}$ other than variables. Indeed one can always apply the Definition rule to separate such terms.

Unification. Applying unification can result in derived equalities between variables of $\mathcal{T}$. These should simply be preserved in the reduced problem.

Dual view. Applying the rewriting into the dual view can result in derived equalities of the form $x = t$, where x is a variable from $\mathcal{T}$ and t is an application of selectors to an input variable. Because $\mathcal{T}$ cannot handle these selectors, we need to rewrite t into a simple variable. By using the Definition and Sequencing rules, we make this variable an input of the problem in $\mathcal{T}$.

Disequalities. Contrary to the pure case, we cannot always eliminate all conjuncts in the CNF by applying a diagonalization; we can eliminate variables that belong to the term algebra, but not variables of $\mathcal{T}$. Instead, for each disequality $v \neq w$ over $\mathcal{T}$ in the CNF, we introduce at the top-level a disjunction $v = w \vee v \neq w$, and apply the Case Split rule to encode a guess. This in essence compiles the guessing of the partitioning of shared variables that is traditionally introduced in a Nelson-Oppen setting [15]. Because Case Split preserves the relation entirely, this is a sound and complete reduction step.

Once all the disequalities have been handled, either through diagonalization if they are over algebraic terms, or by case-splitting if they are over $\mathcal{T}$ variables, we have entirely reduced the synthesis problem into a synthesis problem for $\mathcal{T}$.

5 Synthesis for Arrays with Symbolic Bounds

This section introduces a synthesis procedure for a theory of arrays. In contrast to many other theories for which synthesis procedures have been introduced, the standard (unbounded) array theory does not admit quantifier elimination. With a known finite bound on the size of all arrays, there is a procedure that reduces the synthesis problem to synthesis problems over indices and elements, in a similar way as the satisfiability problem for arrays is reduced to these component theories. However, if we do not know the size bounds at compile time, we need to employ a mixed approach, which postpones some of the reasoning to run time. The reduction is the same as before, but now the component synthesis procedures not only return one solution of the synthesis problem, but instead an iterator over all possible solutions (given by any limited knowledge about the inputs contained in the specification formula). Then, at run time, the synthesized code examines all the solutions for constraints on indices, searching for one that matches the current array inputs.

In the rest of this section we focus on this more general case of symbolic bounds, then revisit briefly statically bounded arrays as a special case.

5.1 Preliminaries

We present synthesis for a *theory of arrays with symbolic bounds*. We consider arrays with the usual read and write operations, an index theory $\mathcal{T}_I$ with an ordered, discrete domain, and an element theory $\mathcal{T}_E$. We assume that our input formula ϕ is a conjunction of literals, and that we have synthesis procedures for these theories. Additionally, we assume that we have a predicate $\approx_I$ between arrays, where I can be any set of variables or constants, and $a \approx_I b$ evaluates to true iff a and b are equal up to (the elements stored at) indices $i \in I$. In particular, this also subsumes extensionality of arrays (with $I = \emptyset$).

Arrays with Symbolic Bounds. We assume that our specification ϕ contains, for every array variable a, two special variables a_l, a_u, standing for the lower and upper bound of the array. Additionally, we assume that ϕ contains, for every

index variable i used to read or write into a, the constraints $a_l \leq i \wedge i \leq a_u$. These constraints ensure that synthesized programs to not contain out-of-bounds array accesses, and that the number of possible solutions for index variables will be bounded at run time. If a is an array parameter, then a_l, a_u are additional parameters. For convenience, if we have $b = \mathsf{write}(a, i, e)$ or $a \approx_I b$, then we assume that $a_l = b_l$ and $a_u = b_u$, i.e. we need not introduce multiple lower and upper bounds for "connected" arrays.

Enumerating Solutions For the index theory $\mathcal{T}_I$, we need a synthesis procedure that not only returns one solution $\overline{T}$, but allows us to iterate over the set $\overline{T}^*$ of all possible solutions. We assume that one of the following cases holds:

1. the synthesis procedure computes $\overline{T}^*$ as a finite set of solutions
2. the synthesis procedure computes $\overline{T}^*$ as a solved form of ϕ, that allows us to access solutions iteratively (like mentioned in Sect. 3; this is also possible if there are infinitely many possible solutions at compile time)
3. the synthesis procedure produces code $\overline{T}^*$, representing a specialized solver for the index theory, that is instantiated with ϕ and allows to add more constraints ψ to obtain solutions satisfying $\phi \wedge \psi$ (in the limit this means integrating a constraint solver procedure into the generated code [7]).

5.2 A Reduction-Based Synthesis Procedure for Arrays

We introduce a synthesis procedure for arrays, consisting of the following steps:

- **Array reduction:** ϕ is reduced to $\mathcal{T}_E \cup \mathcal{T}_I$, along with a set of definitions that allows us to generate witness terms for array variables;
- **"Partial Synthesis" reduction:** part of the reasoning is postponed to runtime, assuming we get an enumerator of all possible solutions in $\mathcal{T}_E \cup \mathcal{T}_I$;
- **Separation and synthesis in $\mathcal{T}_E$ and $\mathcal{T}_I$:** we separate the specification into parts talking purely about $\mathcal{T}_E$ and $\mathcal{T}_I$, and synthesize all possible solutions.

Array Reduction. We introduce fresh variables for array writes and reads, allowing us to reduce the problem to the combined theory $\mathcal{T}_I \cup \mathcal{T}_E$:

1. For every array write $\mathsf{write}(a, i, e)$ in ϕ: i) use Definition to introduce a fresh array variable b, and obtain $b = \mathsf{write}(a, i, e) \wedge \phi[write(a, i, e) \mapsto b]$, and ii) by Equivalence, add $b[i] = e \wedge a \approx_{\{i\}} b$ to ϕ.
2. Until saturation, use Equivalence to add for every pair of literals $a \approx_I b$ and $b \approx_J c$ in ϕ the literal $a \approx_{I \cup J} c$.
3. For every array read $a[i]$ and predicate $a \approx_J b$ in ϕ: use Equivalence to add a formula $(\bigwedge_{j \in J} i \neq j) \rightarrow a[i] = b[i]$ to ϕ.
4. For every array read $a[i]$ in ϕ: i) use Definition to introduce a fresh element variable a_i , and obtain $a_i = a[i] \wedge \phi[a[i] \mapsto a_i]$.
5. For every pair of variables a_i, a_j in ϕ: use Equivalence to add a formula $i = j \rightarrow a_i = a_j$ to ϕ.

Let $D \equiv D_1 \wedge D_2$, where D_1, D_2 are the sets of definitions introduced in 1 and 4, respectively. Let Eq_A be the saturated set of all literals $a \approx_I b$ after 2, Impl the set of all implications introduced in 3 and 5, and ϕ' the remaining part of ϕ, after the rewriting steps in 1 and 5. Let furthermore $\bar{b}, \bar{a_i}$ be the sets of fresh variables introduced in steps 1 and 5, respectively. Then array reduction can be depicted as a macro-step

$$\frac{[\![\bar{a}\ \ \langle \phi' \wedge \mathsf{Impl} \wedge Eq_A \wedge D \rangle\ \ \bar{b}\,;\bar{a_i}\,;\bar{x}]\!] \vdash \langle P \mid \bar{T} \rangle}{[\![\bar{a}\ \ \langle \phi \rangle\ \ \bar{x}]\!] \vdash \langle P \mid \mathsf{let}\ (\bar{b}\,;\bar{a_i}\,;\bar{x}) := \bar{T}\ \mathsf{in}\ \bar{x} \rangle}$$

Let $\bar{x_A}\,;\bar{x_E}\,;\bar{x_I}$ be a separation of $\bar{x}$ into array, element and index variables. As an array-specific step, based on $Eq_A \wedge D$ we can now already give witness terms for array variables $\bar{b}\,;\bar{x_A}$, assuming that we will get witness terms $\bar{T}_{a_i}$ for $\bar{a_i}$ and $\bar{T}_I$ for $\bar{x_I}$:

Let a be an array variable, and I the set of all index variables i for which $a_i = a[i]$ is in D. For a given v, let J be the maximal subset of I s.t. $\forall i, j \in J.\ v \models i \neq j$. By construction, there must be an array variable b s.t. $a \approx_I b$ is in Eq_A. If b is not a parameter array, all positions not explicitly defined can be defined arbitrarily. Then the witness term T_a for variable a is defined by:

$$T_a := \mathsf{write}(\ldots(\mathsf{write}(b, T_{j_1}, T_{a_{j_1}})\ldots), T_{j_n}, T_{a_{j_n}})$$

where the T_j are witness terms for index variables $j \in J$, and the T_{a_j} witness terms for the corresponding element variables. Let $\bar{T_A}$ be the sequence of witness terms for all array variables $\bar{b}\,;\bar{x_A}$. Then this step can be depicted as

$$\frac{[\![\bar{a}\ \ \langle \phi' \wedge \mathsf{Impl} \wedge D_2 \rangle\ \ \bar{a_i}\,;\bar{x_E}\,;\bar{x_I}]\!] \vdash \langle P \mid \bar{T} \rangle}{\begin{array}{l}[\![\bar{a}\ \ \langle \phi' \wedge \mathsf{Impl} \wedge Eq_A \wedge D \rangle\ \ \bar{b}\,;\bar{x_A}\,;\bar{a_i}\,;\bar{x_E}\,;\bar{x_I}]\!] \\ \qquad \vdash \langle P \mid \mathsf{let}\ (\bar{a_i}\,;\bar{x_E}\,;\bar{x_I}) := \bar{T}\ \mathsf{in}\ (\bar{T_A}\,;\bar{a_i}\,;\bar{x_E}\,;\bar{x_I}) \rangle\end{array}}$$

Correctness of this step follows from the correctness of array decision procedures using the same reduction. Note that we also remove Eq_A and D_1 from our specification, as they will not be needed anymore.

Partial Synthesis with Run-Time Checks. Since the theory of arrays does not allow quantifier elimination, we in general need to postpone some of the reasoning to run time. The following is a general rule to separate the specification into a part ϕ that allows for compile-time synthesis, and another part ψ that is checked (against the possible solutions of ϕ) at run time. Here, we assume that the result $\bar{T}^*$ is an iterator over all possible solutions, and that for any given $\bar{a}$ only finitely many solutions exist:

$$\frac{[\![\bar{a}\ \langle\phi\rangle\ \bar{x}]\!] \vdash \langle P \mid \bar{T}^*\rangle}{[\![\bar{a}\ \langle\phi \wedge \psi\rangle\ \bar{x}]\!] \vdash \left\langle P \wedge \exists i.\ \psi[\bar{x} \mapsto \bar{T}^*.\mathsf{next}^{(i)}] \;\middle|\; \begin{array}{l} \bar{x} := \bar{T}^*; \\ \mathsf{while}(\neg\psi(\bar{a}, \bar{x})) \\ \mathsf{if}(\bar{T}^*.\mathsf{hasNext})\ \{\bar{x} := \bar{T}^*.\mathsf{next}\} \\ \mathsf{else}\ \{\mathsf{return\ UNSAT}\} \end{array} \right\rangle}$$

If $\bar{T}^*$ is not an iterator, but a specialized decision procedure for the given theory and constraint ϕ, then the loop is replaced by a call to $\bar{T}^*$ with constraint ψ.

For array synthesis, we apply the rule to remove D_2, reducing the synthesis problem to $[\![\bar{a}\ \langle\phi' \wedge \mathsf{Impl}\rangle\ \bar{a_i}\,;\bar{x_E}\,;\bar{x_I}]\!] \vdash \langle P \mid \bar{T}^*\rangle$, in the theory $\mathcal{T}_E \cup \mathcal{T}_I$.

Separation of $\mathcal{T}_I \cup \mathcal{T}_E$ We use the Sequencing rule to separately synthesize element and index variables:[1]

$$\frac{[\![\bar{a}\,;\bar{x_I}\ \langle\phi' \wedge \mathsf{Impl}\rangle\ \bar{a_i}\,;\bar{x_E}]\!] \vdash \langle P_E \mid \bar{T}_E\rangle \qquad [\![\bar{a}\ \langle P_E\rangle\ \bar{x_I}]\!] \vdash \langle P \mid \bar{T}_I^*\rangle}{[\![\bar{a}\ \langle\phi' \wedge \mathsf{Impl}\rangle\ \bar{x}\,;\bar{y}]\!] \vdash \langle P \mid \mathsf{let}\ \bar{x_I} := \bar{T}_I^*\ \mathsf{in}\ (\bar{T}_E\,;\bar{x_I})\rangle}$$

$\mathcal{T}_E$-Synthesis To solve the left-hand side, note that ϕ' is a conjunction of literals and does not contain array variables anymore, so it can be separated into $\phi_E \wedge \phi_I$, with ϕ_E, ϕ_I pure constraints in $\mathcal{T}_E$ and $\mathcal{T}_I$, respectively. We use the Assertions rule to move ϕ_I into P_E. The only other occurrences of index variables are in Impl. To remove these, we use Equivalence to introduce a disjunction over all possible valuations of equalities between index variables, and Case Split to branch synthesis of element variables for all these cases:

Let Eq_I be the set of all equalities $i = j$ s.t. either $i = j$ or $i \neq j$ appears in an implication in Impl. Let V_E be the set of truth valuations of elements of Eq_I, each described by a conjunction of literals $v \in V_E$ (containing for every $l \in Eq_I$ either l or $\neg l$). For every $v \in V_E$, obtain a new formula ϕ_v by adding to $\phi_E \wedge v$ the succedent of all implications in Impl for which the antecedent is in v.

For each $v \in V_E$, we solve $[\![\bar{a}\,;\bar{x_I}\ \langle\phi_v\rangle\ \bar{a_I}\,;\bar{x_E}]\!] \vdash \langle P_v \mid \bar{T}_v^*\rangle$, where ϕ_v is a pure $\mathcal{T}_E$-constraint. Joining results according to Case Split, we get $\langle P_E \mid \bar{T}^*\rangle$.

$\mathcal{T}_I$-Synthesis We solve the right-hand side of the Sequencing rule, $[\![\bar{a}\ \langle P_E\rangle\ \bar{x_I}]\!] \vdash \langle P \mid \bar{T}_I^*\rangle$. As before we use Assertions to obtain a pure $\mathcal{T}_I$-constraint.[2]

[1] Note that we do not have to explicitly compute all possible solutions in $\mathcal{T}_E$; since $\bar{x_I}$ is used as an input in $\bar{T}_E$, we will obtain a suitable solution of $\bar{a_i}\,;\bar{x_E}$ for every solution of $\bar{x_I}$.

[2] If bounds for all arrays (or all array indices in ϕ) can be computed at compile time, then all solutions can be computed statically. Otherwise, array bounds are symbolic and will only have values at run time, i.e. we need to be able to compute solutions during runtime.

Remarks For efficiency, it may be useful to deduce, both in $\mathcal{T}_I$ and $\mathcal{T}_E$, equalities that are implied by ϕ at any time, and replacing clauses in Impl by their succedents if the antecedent is implied by ϕ, or the negation of the antecedent, if the negation of the succedent is implied. This will avoid unnecessary branching, speeding up synthesis and removing dead branches from the resulting code.

Theory Combination $\mathcal{T}_E \cup \mathcal{T}_I$ The reduction above assumes that theories $\mathcal{T}_E$ and $\mathcal{T}_I$ are *strongly disjoint*, i.e. they share not even the equality symbol. Alternatively, we can make the restriction that variables that are used for array reads may never be compared to variables that are used as elements. In this case, implications from congruence of array reads is the only connection between the theories, and $\mathcal{T}_I$-synthesis can run completely independent of $\mathcal{T}_E$-synthesis, provided the latter accounts for all possible cases of $\mathcal{T}_I$-equalities. If the theories are not strongly disjoint, we really need a synthesis procedure for the combined theory. In this case, we directly use the combined decision procedure to produce an iterator over all possible solutions in both element and index theory.

Statically Bounded Arrays If all arrays in ϕ are statically bounded, i.e. values of upper and lower bounds are known or can be computed at compile time, then we can statically compute all solutions for constraints in $\mathcal{T}_I$ that are within array bounds. In that case the generated code does not need an iterator that computes additional solutions, and we can give a constant bound on the maximum number of traversals of the loop at compile time.

5.3 Complexity of Synthesis and Synthesized Code

Complexity of the array synthesis procedure is dominated by the branching on equalities of index variables: we may need exponentially many (in the number of index variables) calls to the synthesis procedure for $\mathcal{T}_E$. The array reduction itself runs in polynomial time.

Correspondingly, the size of the synthesized code is also exponential in the number of index variables. The code can contain branches for all possible cases of equalities (arrangements) among indices. Although only one of these branches will be explored at runtime, the worst-case running time of the synthesized code will still be exponential in the number of index variables: for a size bound n on a given array, there may be n^i many solutions to the constraints in $\mathcal{T}_I$. In the worst case, the condition of the while-loop needs to be checked for all of these solutions.

5.4 Example of Array Synthesis

Suppose we want to synthesize a most general precondition P and program code s.t. for any input array a and bounds a_l, a_u that satisfy P, the synthesized code

computes values for an array b and integer variables i, j, k such that the following is satisfied:[3]

$$\begin{aligned}\phi \equiv \quad & a_l = 0 \wedge i > a_l \wedge i = j + j \wedge i < a_u \wedge k \geq a_l \wedge k < i \\ & \wedge\, a[i] > 0 \wedge a[k] \leq 0 \wedge b[i] > a[i-2] \wedge b[k] = a[i] \\ & \wedge\, a' = \mathsf{write}(a, i, e_1) \wedge b = \mathsf{write}(a', k, e_2).\end{aligned}$$

Array reduction: We obtain

$$\begin{aligned}D_1 :=\ & \{a' = \mathsf{write}(a, i, e_1), b = \mathsf{write}(a', k, e_2)\} \\ Eq_A :=\ & \{a' \approx_i a, b \approx_k a', a \approx_{\{i,k\}} b\} \\ \mathsf{Impl} =\ & \left\{\begin{array}{llll} k \neq i \ \rightarrow\ a_k = a'_k, & i-2 \neq i \ \rightarrow\ a_{i-2} = a'_{i-2}, \\ i \neq k \ \rightarrow\ a'_i = b_i, & i-2 \neq k \ \rightarrow\ a'_{i-2} = b_{i-2}, \\ i-2 \neq i \wedge i-2 \neq k \ \rightarrow\ a_{i-2} = b_{i-2} & \\ i = k \ \rightarrow\ a_i = a_k, & i-2 = k \ \rightarrow\ a_{i-2} = a_k, \\ i = k \ \rightarrow\ a'_i = a'_k, & i-2 = k \ \rightarrow\ a'_{i-2} = a'_k, \\ i = k \ \rightarrow\ b_i = b_k, & i-2 = k \ \rightarrow\ b_{i-2} = b_k \end{array}\right\}, \\ D_2 :=\ & \left\{\begin{array}{l} a_i = a[i], a_{i-2} = a[i-2], a_k = a[k], \\ a'_i = a'[i], a'_{i-2} = a'[i-2], a'_k = a'[k], \\ b_i = b[i], b_{i-2} = b[i-2], b_k = b[k] \end{array}\right\} \\ \phi' :=\ & a_l = 0 \wedge i > a_l \wedge i = j + j \wedge i < a_u \wedge k \geq a_l \wedge k < i \\ & \wedge\, a_i > 0 \wedge a_k \leq 0 \wedge b_i > a_{i-2} \wedge b_k = a_i \\ & \wedge\, a'_i = e_1 \wedge b_k = e_2\end{aligned}$$

Implied equalities and disequalities: From ϕ' we can conclude that $i \neq j, k \neq i$ and $i - 2 \neq i$, as well as $a_i \neq a_k, b_i \neq a_{i-2}$.

Propagating equalities through Impl*:* $k \neq i$ implies $a_k = a'_k$ and $a'_i = b_i$. $i - 2 \neq i$ implies $a_{i-2} = a'_{i-2}$. In the opposite direction, $a_i \neq a_k$ implies $i \neq k$ (which we already knew). We get

$$\phi'' := \phi' \wedge a_i \neq a_k \wedge b_i \neq a_{i-2} \wedge a'_i = b_i \wedge a_{i-2} = a'_{i-2} \wedge i \neq k \wedge i - 2 \neq i.$$

Separation of $\mathcal{T}_I \cup \mathcal{T}_E$ We use the $\mathsf{Sequencing}$ rule, obtaining subproblems $[\![\bar{a}\,; \bar{x_I}\ \ \langle \phi'' \wedge \mathsf{Impl} \rangle\ \ \bar{a_i}\,; \bar{x_E}]\!] \vdash \langle P_E \mid \bar{T}_E \rangle$ and $[\![\bar{a}\ \ \langle P_E \rangle\ \ \bar{x_I}]\!] \vdash \langle P \mid \bar{T}_I^* \rangle$.

$\mathcal{T}_E$-Synthesis From the three equations that appear in antecedents of Impl, valuations for two are fixed by ϕ''. Thus, we only branch on the valuation of $i - 2 = k$. Let $v_1 \equiv i - 2 = k$, which implies $a_{i-2} = a_k, a'_{i-2} = a'_k$ and $b_{i-2} = b_k$. Let $v_2 \equiv i - 2 \neq k$, which implies $a'_{i-2} = b_{i-2}$ and (together with $i - 2 \neq i$) $a_{i-2} = b_{i-2}$.

[3] Note that the last two literals imply $a \approx_{\{i,k\}} b$, which in turn implies that there exist valuations for a', e_1, e_2 satisfying these literals. Thus, we can allow statements of the form $a \approx_I b$ in specifications, and replace them with a number of write definitions according to the size of I, with fresh element and array variables in every write.

Assuming v_1, we obtain the following valuations for variables $\bar{x}_E$:

$$e_2 := b_k := b_{i-2} := a_i, e_1 := b_i := a'_i := a_{i-2} + 1, a'_k := a'_{i-2} := a_k.$$

Assuming v_2, valuations are the same except for $b_{i-2} := a'_{i-2} := a_{i-2}$. The precondition is in both cases $P_E \equiv a_i > 0 \wedge a_k \leq 0$ (plus the $\mathcal{T}_I$-part of ϕ'').

$\mathcal{T}_I$-Synthesis We obtain $j := \lfloor \frac{i}{2} \rfloor$ and an iterator $\bar{T}_I^*$ of solutions for (i, k):

```
T*_I := (0,2)
T*_I.next = let (k,i) = T*_I in
                    if(k+1<i) (k+1,i)
                    else if(i+2<a_u) (0,i+2)
                    else return UNSAT
```

along with a precondition $P \equiv a_i > 0 \wedge a_k \leq 0 \wedge a_u > 2$.

Array synthesis: Lifting the witness terms for elements to array b, we obtain

$$\begin{array}{l} b := \mathsf{if}(i-2=k) \\ \quad \mathsf{write}(\mathsf{write}(a, i, a[T_{i-2}]+1), T_k, a[T_i]) \\ \quad \mathsf{else}\ \mathsf{write}(\mathsf{write}(\mathsf{write}(a, i, a[T_{i-2}]+1), T_{i-2}, a[T_i]), T_k, a[T_i]) \end{array}$$

Result: Finally, we obtain the precondition

$$a_u > 2 \wedge \exists n.\ \left((i,k) = T_I^*.\mathsf{next}^{(n)} \ \rightarrow\ a[i] > 0 \wedge a[k] \leq 0\right)$$

and the program[4] in Fig. 1 for computing i, j, k and b from a and a_u.

5.5 Example: Inverting Program Fragments

The synthesis procedure for arrays can also be used to invert given code fragments, e.g. for automatically obtaining a program that reverts (some of) the changes a given piece of code did to some data. Consider the code fragment

```
if(a[i]==0)
        a[i]:=a[i+1]
else if (a[i]==1)
        a[i]:=a[0]
else if (a[i]>1)
        a[i]:=a[i]−1
else a[i]:= a[i]+2
```

which translates into the specification

$$\begin{array}{r} (a_0[i] = 0 \wedge a_1 = \mathsf{write}(a_0, i, a_0[i+1])) \\ \vee\ (a_0[i] = 1 \wedge a_1 = \mathsf{write}(a_0, i, a_0[0])) \\ \vee\ (a_0[i] > 1 \wedge a_1 = \mathsf{write}(a_0, i, a_0[i]-1)) \\ \vee\ (a_0[i] < 0 \wedge a_1 = \mathsf{write}(a_0, i, a_0[i]+2)), \end{array}$$

[4] The code can be significantly simplified by merging parts that are not affected by case distinctions.

```
if(a_u > 2) {
        (i,k) := T_I^* in
        while (¬(a[i] > 0 && a[k] ≤ 0))
                if (T_I^*.hasNext)
                        (i,k) := T_I^*.next
                else throw new Exception("Unsatisfiable constraint.")
        let j = i / 2 in
        if (i−2 = k) {
                bi := a[i−2]+1
                bk := a[i]
        } else {
                bi := a[i−2]+1
                bk := a[i]
                bi2 := a[i−2]
        }
        if (i−2 = k)
            b := write(write(write(a,i,bi),k,bk))
        else
            b := write(write(write(a,i,bi),k,bk),i−2,bi2)
        (b,i,j,k)
} else throw new Exception("Unsatisfiable constraint.")
```

Fig. 1. Example of code generated by array synthesis procedure

where a_0 refers to the pre-, and a_1 to the post-state value of array a. For synthesizing the inverted code, we assume that a_1 is the input, and a_0 the output. The synthesis procedure will return a piece of code

```
if(a[i]==a[i+1])
        a[i]:=0
else if(a[i]=a[0])
        a[i]:=1
else if(a[i]>0)
        a[i]:=a[i]+1
else a[i]:=a[i]−2
```

Since the relation given by the input code does not model a bijection, applying the inverted code after the input code will not result in exactly the same state. However, for a deterministic code, the resulting state will be equivalent with respect to the original piece of code: if we run the original program for the second time from such state, we will get the same final result as when running the program once.

6 Related Work

Term algebras admit quantifier elimination [5, 21] and are thus natural candidates for synthesis. Our synthesis procedure is similar to quantifier elimination

when it comes to eliminating variables that are constrained by an equality, with the additional requirement that the witness term be stored to serve in the program. However, we simplified the treatment of disequalities: existing elimination procedures typically rewrite a disequality between a variable and a term into a disjunction of equalities between the same variable and terms constructed with different constructors [5, p.63sqq]. This has the advantage that the language of formulas needs not be extended, allowing for nested quantifiers to be eliminated one after the other. In our synthesis setting, this is not necessary: we can rely on additional computable functions, as we have illustrated with the use of Δ, greatly simplifying the resulting program. A related area of research is compilation of unification in Prolog [1]. This process typically does not require handling of disequalities, so it deals with a simpler language.

Pattern-matching compilation is a task for which specialized procedures for term algebras have been developed [13, 23]. When viewed through the prism of synthesis procedures, these algorithms can be thought of as procedures that are specialized for disjunctions of term equalities, and where the emphasis is put on code reuse. We expect that using a combination of our synthesis procedures and common subexpression elimination techniques, one should be able to derive pattern-matching compilation schemes that would support, e.g., disjunctive patterns, non-linear patterns, and could take into account guards referring to integer predicates.

Our synthesis procedure for arrays is based on a reduction of constraints over arrays to constraints in the combined theory of indices and elements. In particular, our reduction is very close to the decision procedure for extensional arrays introduced by Stump et al. [20]. Combination of strongly disjoint theories is also used in the array decision procedure of de Moura and Bjørner [14], but the main focus of their work was to make array decision procedures more efficient by restricted application of fine-grained reduction rules. In the presence of unknown inputs, these techniques are not applicable in general.

Specialization of decision procedures for the purpose of predicate abstraction was considered in [10]. In addition to covering a different set of theories, our results are broader because our process generates not only a satisfiability check but also the values of variables.

7 Conclusions

We presented synthesis procedures for two important theories: algebraic data types and arrays, formulated in a unified framework. Our contribution fills an unexplored area between two approaches: running SMT solvers at run time [7] and using quantifier-elimination-like algorithms to compile specifications. In this paper we have shown that for two important theories supported by SMT solvers, compilation of constraints is also feasible. Moreover, much like SMT can be built and proved correct modularly, synthesis procedures can be combined as well using our framework.

References

1. Aït-Kaci, H.: Warren's Abstract Machine: A Tutorial Reconstruction. MIT Press (1991) (available online as PDF)
2. Baader, F., Snyder, W.: Unification theory. In: Handbook of Automated Reasoning. Elsevier (2001)
3. Flener, P.: Logic Program Synthesis from Incomplete Information. Kluwer Academic Publishers (1995)
4. Hamza, J., Jobstmann, B., Kuncak, V.: Synthesis for regular specifications over unbounded domains. In: FMCAD, pp. 101–109 (2010)
5. Hodges, W.: A Shorter Model Theory. Cambridge University Press (1997)
6. Jobstmann, B., Bloem, R.: Optimizations for ltl synthesis. In: FMCAD, pp. 117–124 (2006)
7. Köksal, A.S., Kuncak, V., Suter, P.: Constraints as control. In: POPL, pp. 151–164 (2012)
8. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Software synthesis procedures. CACM 55(2), 103–111 (2012)
9. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: PLDI, pp. 316–329 (2010)
10. Lahiri, S.K., Ball, T., Cook, B.: Predicate Abstraction via Symbolic Decision Procedures. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 24–38. Springer, Heidelberg (2005)
11. Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. CACM 14(3), 151–165 (1971)
12. Manna, Z., Waldinger, R.J.: A deductive approach to program synthesis. TOPLAS 2(1), 90–121 (1980)
13. Maranget, L.: Compiling pattern matching to good decision trees. In: ML, pp. 35–46 (2008)
14. de Moura, L.M., Bjørner, N.: Generalized, efficient array decision procedures. In: FMCAD, pp. 45–52 (2009)
15. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. TOPLAS 1(2), 245–257 (1979)
16. Smith, D.R.: KIDS: A semiautomatic program development system. TSE 16(9), 1024–1043 (1990)
17. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS, pp. 404–415 (2006)
18. Spielmann, A., Kuncak, V.: Synthesis for unbounded bit-vector arithmetic. In: IJCAR, pp. 499–513 (2012)
19. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: POPL, pp. 313–326 (2010)
20. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for an extensional theory of arrays. In: LICS, pp. 29–37 (2001)
21. Sturm, T., Weispfenning, V.: Quantifier elimination in term algebras: The case of finite languages. In: Ganzha, V.G., Mayr, E.W., Vorozhtsov, E.V. (eds.) Computer Algebra in Scientific Computing (CASC). TUM München (2002)
22. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: POPL, pp. 327–338 (2010)
23. Wadler, P.: The Implementation of Functional Programming Languages: Efficient Compilation of Pattern-matching, ch. 5, pp. 78–103 (1987)